Exploring Length Generalization in Small Transformers

Ramgopal Venkateswaran
ram1998@stanford.edu

1 Extended Abstract

Motivated by understanding the generalization and reasoning capability of transformers on simple algorithmic tasks, we study three directions related to the ability of small transformers to length generalize:

1.1 Multi-Task Length Generalization

Zhou et al. [2023] conjecture that a particular subset of programs (RASP-L programs) are amenable for small transformers to learn with strong length generalization. The backbone of this class of programs is a function primitive that acts like a transformer block on a (key, query, value) sequence. They show strong length generalization results for problems in this class, and weak generalization for simple problems that fall outside this class.

We take three tasks in this subset which exhibit length generalization in the single-task setting: counting, mode, and parity with scratchpad. We extend them to the multi-task setting in two different ways, and compare the generalization of these methods to their respective single-task baselines. In general, we observe **successful length generalization in the multi-task setting**, with **a small amount of negative interference depending on the encoding we use for the multi-task setting**.

1.2 RASP-L vs Regular Languages

Learning regular languages in a length-generalizable way is generally considered challenging for transformers (unlike RNNs) - ways to get around this have generally involved some changes to the Transformer architecture (Liu et al. [2023], Press et al. [2022]). However, Zhou et al. [2023] showed significantly improved generalization for parity, traditionally considered a very hard task, using a combination of a scratchpad and index hints.

We extend the idea there to show that **regular languages are a subset of RASP-L when the transformer is allowed to train with a scratchpad and index hints**. We combine this with the conjecture in Zhou et al. [2023] to **conjecture that regular languages can be learned in a length-generalizable way by transformers, via a scratchpad and index hints**. We provide experimental evidence for this by training a model that length generalizes on arithmetic mod 5 (including +, - and x) using index hints and a scratchpad; to the best of our knowledge, previous results have required changes to the transformer architecture to demonstrate this length generalization.

1.3 Connections to In-Context Learning

Given that the hypothesis for why index hints work so well is that they facilitate the formation of "induction heads" within the transformer, which Olsson et al. [2022] showed are critical to in-context learning, we look to understand if we can leverage index hints to facilitate in-context learning as well. We show that **index hints can facilitate length generalization in in-context learning for the "copy with repeats" task**.

2 Introduction

2.1 Motivation

The transformer architecture, first introduced in Vaswani et al. [2023], has since demonstrated widespread success in natural language processing tasks, particularly through the use of large language models (LLMs) trained on large text corpora (Brown et al. [2020], Touvron et al. [2023]). They have also been successfully applied to other domains such as vision (Dosovitskiy et al. [2021]).

An interesting facet of LLM performance is that they can be prompted to achieve good performance on downstream tasks involving algorithmic reasoning, such as on high-school-level examinations (e.g. OpenAI [2023]) and generating code to solve problems (e.g. Rozière et al. [2023]). On the other hand, such models can also fail in unintuitive ways (Lacker [2020]), which motivates the question of what the extent of their reasoning capabilities is, and what are the correlations in data are they able to learn and generalize based on?

One approach to making progress on this question is to look at small transformers in controlled settings with well-defined tasks, instead of large transformers trained on large and diverse datasets (whose behavior is harder to reason about). For this purpose, we can also employ a more precise definition of what it means to have learned how to solve a simple algorithmic problem - recent lines of work have looked at the notion of **length generalization**, which is the ability of a model to generalize well from the algorithmic problems it has seen in its training data (in-distribution) to that same class of algorithmic problems but with longer inputs (out-of-distribution).

Based on the above considerations, we then arrive at the question of understanding the length generalization capabilities of small transformers (specifically, in this work, all models will have at most 20 million parameters) on simple algorithmic reasoning tasks.

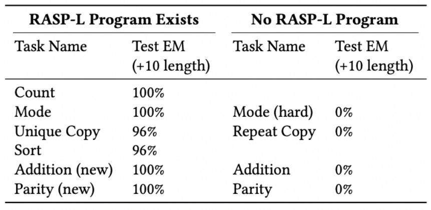
2.2 Related Work

2.2.1 Length Generalization with RASP-L

Zhou et al. [2023] recently studied the length generalization question for simple algorithmic tasks such as counting, parity, and addition. They conjectured that a subset of "RASP programs" - a simple class of programs first introduced in Weiss et al. [2021] - are amenable for small transformers to learn with strong length generalization. They build upon other work including Lee et al. [2023] and Awasthi and Gupta [2023], demonstrating strong length generalization capabilities on a series of tasks within this set of problems. This included the first instance of length generalization on some tasks that were previously considered hard for transformers such as parity and addition (after making some changes to the distribution and input format).

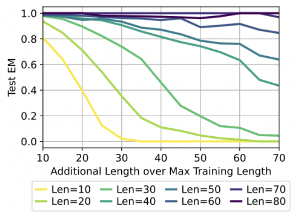
"RASP" is a simple language whose function primitives are designed to model the workings of a transformer's attention block - its key building block is an operation that takes in a binary predicate, a key sequence, a value sequence, and a query sequence, creates an attention matrix from applying the binary predicate to all (key, query) pairs, and then applies the matrix to the value sequence (refer to Weiss et al. [2021], Lindner et al. [2023], and Zhou et al. [2023] for more detailed versions of this explanation). Lindner et al. [2023] developed a library to compile RASP-L programs to transformers whose weights are designed to solve the simple algorithms specified by their RASP program.

Zhou et al. [2023] then considered the question of which among these RASP programs does a transformer end up succesfully learning in practice, and introduced two more conditions to generate a variant of RASP known as "RASP-L" - one of these conditions restricts RASP to not allow arbitrary indexing operations (and instead only allowing order comparison and predecessor/successor operations), and the other mildly extends RASP to include "min" and "max" aggregations (with the motivation behind these restrictions explained in the paper). They then examine the length generalization capabilities of a set of tasks in and not in RASP-L - for some tasks that are not in RASP-L, they show that adding certain scratchpads and index hints can convert it into a RASP-L program, and demonstrate that transformers can length-generalize on these programs and not on their non-RASP-L counterparts - this is demonstrated in figure 1 of their paper, which we reproduce as figure 1 here. Other interesting facets of the work include showing that certain intuitive scratchpads that are not in RASP-L do not exhibit length generalization as well as their RASP-L counterparts, and comparisons between RASP-L and other suggested notions of transformer learning like min-degree-interpolators from Abbe et al. [2023].



RASP-L Program Exists		No RASP-L Program	
Task Name	Test EM (+10 length)	Task Name	Test EM (+10 length)
Count	100%		
Mode	100%	Mode (hard)	0%
Unique Copy	96%	Repeat Copy	0%
Sort	96%		
Addition (new)	100%	Addition	0%
Parity (new)	100%	Parity	0%

(a) Transformer length generalization



(b) Length generalization on counting task

Figure 1: This figure is taken from Zhou et al. [2023].

2.3 Other Approaches to Length Generalization

Other approaches to length generalization have also been studied - Delétang et al. [2023] showed that while RNNs performed well on a set of regular languages including parity and modular arithmetic, transformers had poorer length generalization, particularly on tasks which were not permutation-invariant. Work by Press et al. [2022] modified the attention mechanism to include recency biases and Liu et al. [2023] combined that with the idea of using a scratchpad (Nye et al. [2022]) to show good length generalization on natural language tasks. Recently, Chi et al. [2023] created a variant of the transformer architecture with a working memory (RegularGPT) which showed successful generalization on regular languages, without requiring additional modifications like a scratchpad.

2.3.1 Connections to In-Context Learning

Zhou et al. [2023] also show that "induction heads" can be implemented as a short program in RASP-L and conjecture that tasks that can utilize these induction heads are particularly easy for transformers to length generalize on. Induction heads were first introduced in Olsson et al. [2022], which found strong evidence for induction heads contributing to a majority of in-context learning especially in smaller models. Induction heads refer to an instance of a model predicting what token will come next after the current token, based on what token came next after the previous instance of the current token.

2.4 Objectives

We look to extend the related work discussed above by exploring the following three questions:

1. Given a set of individual tasks that length generalize well (from Zhou et al. [2023]), can we demonstrate length generalization across multiple tasks trained together?
2. Can we generalize the methods used in Zhou et al. [2023] to improve generalization for parity to other regular languages in general, connecting it with the line of work on improving transformer performance on regular languages?
3. We used "index hints" to boost length generalization by converting non-RASP-L tasks to RASP-L tasks. Based on the connection to induction heads (which are known to be important for in-context learning), can we use the same idea to boost learning and generalization in the in-context learning setting?

3 Multi-Task Length Generalization

3.1 Methods

Our first goal is to understand the ability of small transformers to length generalize in the multi-task setting - we take three tasks which exhibit length generalization in the single-task setting: counting, mode, and parity with scratchpad. We provide toy examples of each of these tasks in 2 to illustrate the goal of each task as well as what length generalization refers to in each case. The detailed specification of each task is exactly as outlined in Appendix B of Zhou et al. [2023]. Note that "parity with scratchpad" is an illustration of a language that would not be in RASP-L without the scratchpad but is in RASP-L and shown to be length generalizable with scratchpads (and index hints) in the single-task setting.

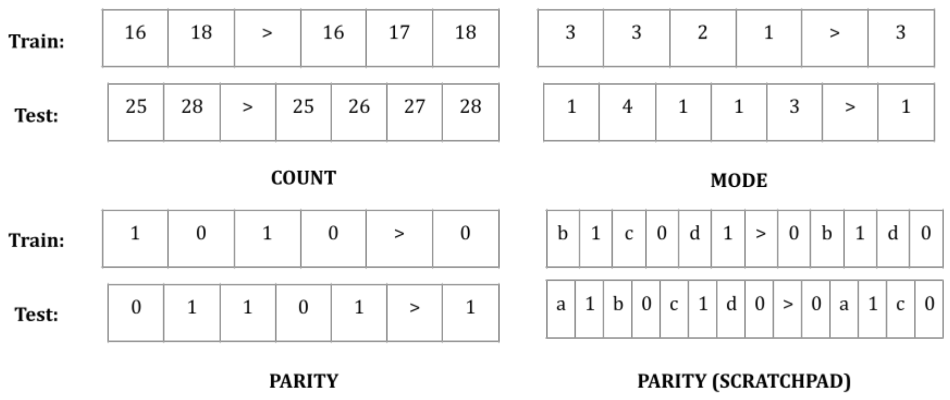


Figure 2: Toy length generalization examples for each of the tasks we will consider in this section. "Train" refers to a smaller length example that we might train on and "Test" refers to a longer example that we might test on to assess generalizability (Note: the true lengths we train/test on will be much larger.)

3.2 Results & Analysis

We first reproduce the baselines from the paper for each task in the single-task setting, and verify that we can show length generalization in all cases. We use similar causal decoder-only transformer architectures to what was outlined in the paper as well as similar training methods - we pack and shift the context, use learned positional embeddings, and tokenize each number individually. There are only minor differences in training and architecture such as batch size, number of layers, etc. in order to satisfy computational constraints (GPU memory) - the details of all architectures we found are in table 1 as well as the submitted code (link: https://github.com/ramvenkat98/nanoGPT-length-generalization/).

Task	Model Size	Train Iter	Context Len	Batch Size	Learning Rate
Count	6 layer; 8 head; 64 emb	10000	256	128	1e-3 to 1e-4
Mode	6 layer; 8 head; 512 emb	10000	256	128	1e-3 to 1e-4
Parity	6 layer; 8 head; 256 emb	10000	512	128	1e-3 to 1e-6
Multi-Task	6 layer; 8 head; 512 emb	10000	512	64	1e-3 to 1e-4
Arithmetic Mod 5	6 layer; 8 head; 128 emb	10000	512	64	1e-3 to 1e-4

Table 1: Architecture and training parameters for each of the transformer models that we train; compare to Zhou et al. [2023] for reference.

In addition to the single-task baselines for each of these three tasks, we also train multi-task models in two ways:

1. **No overlapping encoding**: The first way is to expand our vocab size to accommodate the multi-task case by making the vocabulary of each task disjoint from the vocabulary of other tasks. That is, if our vocabulary size for the three tasks was $|V_{\texttt{count}}|$, $|V_{\texttt{mode}}|$, $|V_{\texttt{parity}}|$, our new vocabulary size would be $|V_{\texttt{count}}| + |V_{\texttt{mode}}| + |V_{\texttt{parity}}|$, with each token only appearing in one of the given tasks and no common tokens. Note that this is **not scalable** but fully prevents negative interference between tasks, allowing for a comparison with the second approach.

2. **Overlapping encoding**: The second way is to allow all vocabularies to overlap, but adding a special token at the start of each task sequence indicating which task we are in. That is, the new vocabulary size would just be $\max(|V_{\texttt{count}}|, |V_{\texttt{mode}}|, |V_{\texttt{parity}}|) + 3$.

We train on both the ways mentioned above in a 2-task case (just count and parity) as well as the 3-task case (count, parity, and mode) in order to understand if there is any clear drop-off in existing tasks' performance with the introduction of a new task. Architectures are similar to the single task case and detailed in 1 as before. In the multi-task case, we sample a task uniformly at random for each batch; in both the single and multi-task cases, we sample the lengths within each batch randomly between 1 and 50 at training time. At evaluation time, we evaluate on lengths from 50 to 100 (inclusive) at intervals of 10 with 20 evaluation samples per interval.

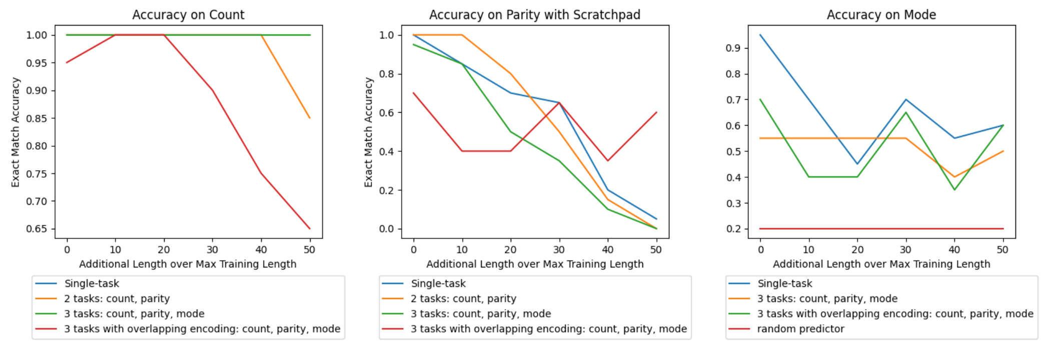


Figure 3: MTML Generalization Results Compared to Single Task Baselines. For the two plots on the left, a random predictor would get roughly 0 accuracy because it would have to predict a long sequence of tokens exactly - for the plot on the right, a random predictor would get 0.2 accuracy since it needs to only predict a single token and has a 1 in 5 chance because we use 5 tokens for mode.

We can observe and conclude the following from the results:

1. **Baseline Single-Task Performance**: For all tasks, the baseline single-task performance that we obtain generally matches the results in Zhou et al. [2023] and are within expectations. For counting, it is an exact match (strong generalization, with 100% accuracy even up to length 100 when training on length 50). For mode, our performance is slightly worse (comparing to the `length=50` plot in figure 3a in the paper) but still in a similar range, ending at around 0.6 for length 100. For parity, we don't have a direct comparison of training up to length 50 in the paper but our results lie in the expected range when compared to the other plots showing training up to length 45.

2. **3 Tasks, Multi-Task Performance without Overlapping Encoding**: We see good generalization in the multi-task case without overlapping encoding, with all tasks having nearly comparable performance to their respective single-task scenarios. For count, the accuracy remained at 100% throughout, and for other two, while the accuracy was generally lower, it was not significantly so.

3. **3 Tasks, Multi-Task Performance with Overlapping Encoding**: With overlapping encoding, we interestingly see accuracy on count drop and no longer be consistently at 100% throughout, which suggests that there is some negative interference. For other tasks, we see similar performance to the non-overlapping encoding case (with even better performance than other cases at high eval lengths for parity with scratchpad).

4. **Two tasks versus three tasks**: The two task case is generally similar to the three task case, with one exception being that the two task case does not get 100% accuracy at length 100 for count - however, this may also be noisy due to the low number of eval samples at each length (20) and we don't see this be a consistent pattern at other lengths and tasks.

In general, we observe **successful length generalization in the multi-task setting**, with **a small amount of negative interference on the 'COUNT' task when overlapping encodings are used**.

4 Regular Languages vs RASP-L

4.1 Methods

We next aim to extend the length generalization result obtained by Zhou et al. [2023] to all regular languages.

To the best of our knowledge, previous work has shown the following two things with regards to the ability of transformers to length generalize on regular languages:

1. When augmented with a working memory, transformers can length generalize on regular languages without a scratchpad as shown in Chi et al. [2023].

2. When the attention mechanism of a transformer is modified to add a recency bias to it (that biases it towards putting more attention weight on recent inputs) and it is trained with a scratchpad, transformers can length generalize to regular languages as shown in Liu et al. [2023].

Applying the same ideas from Liu et al. [2023] and the "parity" case in Zhou et al. [2023] gives us the fact that **regular languages are a subset of RASP-L when the transformer is allowed to train with a scratchpad and index hints**. We can combine this with the conjecture in Zhou et al. [2023] that RASP-L programs are learnable by transformers in a length generalizable way, to **conjecture that regular languages can be learned in a length-generalizable way by a transformer, via a scratchpad and index hints**.

We first clarify what an index hint is: adding an index hint means to interleave each input with an index position - take the "test" example for the "parity (scratchpad)" task in figure 2 for example:the regular parity input sequence there would be $(1, 0, 1, 0)$ and the expected output would be 0. Adding index hints involves transforming this into $a, 1, b, 0, c, 1, d, 0$ - that is, we interleave the inputs with a unique token identifying the position of the input. When combined with a scratchpad, we can then expect an output of $0, a, 1, c, 0$ where the scratchpad records a 0 initially and then records the index position each time it sees a 1 and flips the parity bit accordingly.

As mentioned in Zhou et al. [2023], an index hint is useful because it allows the transformer to record the full current state of the program externally (rather than storing it within its attention mechanism) - specifically for parity, the transformer need not perform complex index arithmetic to find its previous state (i.e. which token of the input it should look at next to update the parity bit) - it can use an induction head (described in the section above) to look up the previous occurrence of the index hint that has been recorded to the output and then attend to that.

This would apply not just for parity but for any regular language. To show this, we note that regular languages are equivalent to the set of languages that can be recognized by a discrete finite automaton (DFA), which we can represent as $(Q, \Sigma, \delta, q_0, F)$. We will equivalently write the problem as identifying the correct final q_T (where T is the length of the input) instead of outputting 1 or 0 based on whether the final state $q \in F$.

Liu et al. [2023] introduced the idea of using the intermediate states as the scratchpad - they interleave the current state q_i with the input bits σ_i - i.e. the scratchpad will be of the state $(\sigma_1, q_1, \sigma_2, q_2, \cdots, \sigma_T, q_T)$ where q_i is the state after the sequence $(\sigma_1, \cdots, \sigma_i)$. They note that a recency bias is also needed in addition to this so that the model does not learn a wrong parallel "shortcut" to the problem.

In our case, instead of interleaving the current state with the corresponding input bit, we interleave it with a unique index hint - this is illustrated in figure 4. Then the corresponding problem would be in RASP-L (because we no longer have any non-trivial indexing operations) and the transformer can leverage an induction head to learn to focus on the previous occurrence of the latest index.

To obtain evidence for the conjecture, we look to learn a more complicated regular language (i.e. more states, transitions) than parity using the same method: modular arithmetic mod 5 (+, -, and x) from Chi et al. [2023].

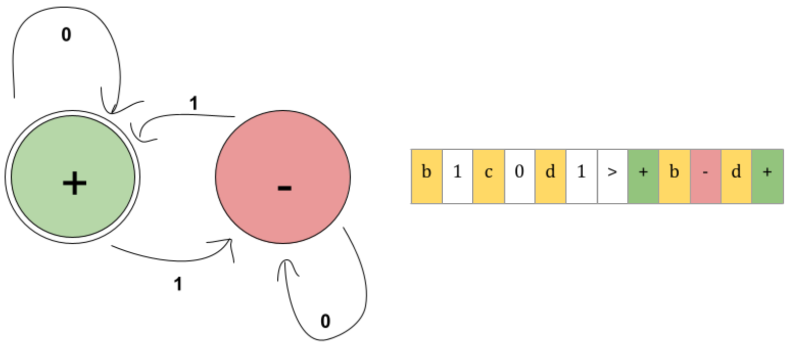


Figure 4: Illustration of the mapping between the DFA for a regular language and the corresponding scratchpad and index hints; states are color-coded in green and red, index hints are in orange.

4.2 Results & Analysis

We experimentally observe **length generalization on mod 5 with scratchpad and index hints** - the strength of the additional generalization also seems to increase with our max train length (e.g. training up to sequences of length 80 yields 70% accuracy even at double the length, unlike training to lengths 30 and 50 (see figure 5). This is a datapoint that supports our conjecture of transformers being able to learn regular languages in a length generalizable way, without any modifications to the architecture, via scratchpads and index hints. (At the same time, this is only a single instance of a regular language - further work would be needed to understand if this holds true in general.)

Note that the model architectures, training procedures, etc. are similar to previously described tasks and the specifications can be found in 1. The scratchpad that we use comprises a standard DFA for mod 5 arithmetic with the state encoding the current result and transitions depending on the next symbol and quantity.

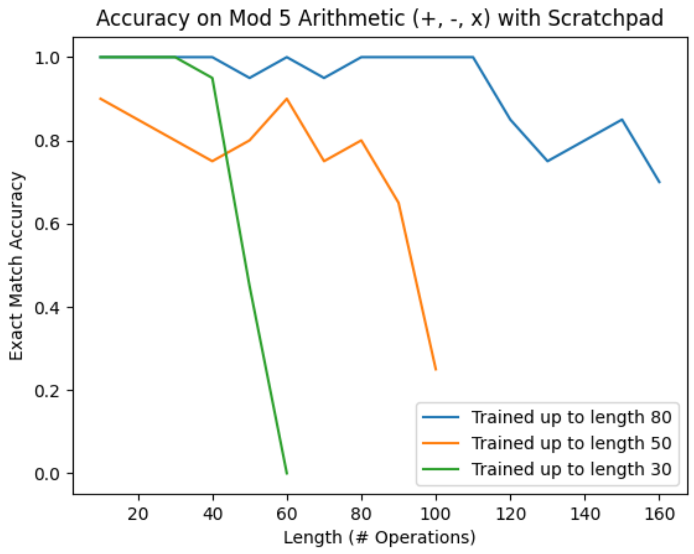


Figure 5: Length generalization capabilities, as a function of max length that the transformer was trained on, for the mod 5 arithmetic task. Note that a random predictor would have an exact match accuracy of around 0.0, not 0.2, because this tests for exact match of the whole scratchpad sequence.

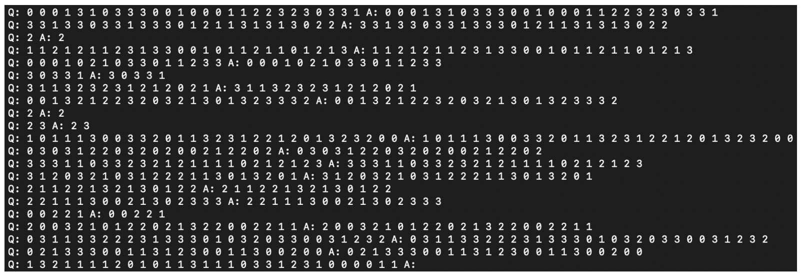
5 In-Context Learning with Induction Heads

5.1 Methods

Given that the hypothesis for why index hints work so well is that they facilitate the formation of "induction heads" within the transformer, which Olsson et al. [2022] showed are critical to in-context learning, we look to understand if we can leverage index hints to facilitate in-context learning as well.

Specifically, we test this hypothesis on the "copy with repeats" task, which is a simple task requiring the transformer to copy its input as output, where the input may include repeat tokens. This task does not have a short RASP-L program and the length generalization on this task is much worse than the "copy (unique)" task (where we look to copy a sequence with unique tokens) as mentioned in Zhou et al. [2023].

We modify the problem scope as follows - we provide a set of in-context samples (fixed at 20 samples) with each sample having a length randomly sampled between 1 and 30. We then evaluate the response on a sample of a fixed length (we try evaluating on lengths of 30, 50, 70, or 90). To control the number of repeats, we vary the alphabet size from which we draw the tokens in the input sequence - we vary this from 64 to 2 by powers of 2. With a smaller alphabet size, the proportion of repeats per sample increases and the task becomes harder. An example input can be seen in figure 6.

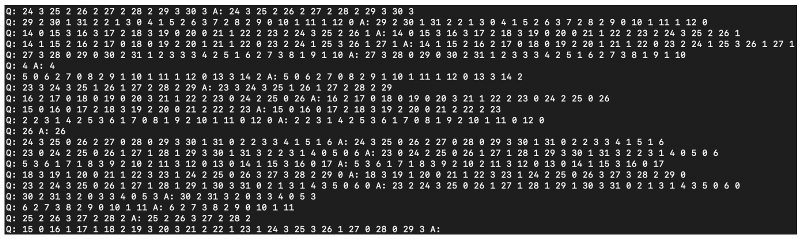


```
Q: 0 0 0 1 3 1 0 3 3 3 0 0 1 0 0 0 1 1 2 2 3 2 3 0 3 3 1 A: 0 0 0 1 3 1 0 3 3 3 0 0 1 0 0 0 1 1 2 2 3 2 3 0 3 3 1
Q: 3 3 1 3 3 0 3 3 1 3 3 3 0 1 2 1 1 3 1 3 1 3 0 2 2 A: 3 3 1 3 3 0 3 3 1 3 3 3 0 1 2 1 1 3 1 3 1 3 0 2 2
Q: 2 A: 2
Q: 1 1 2 1 2 1 1 2 3 1 3 3 0 0 1 0 1 1 2 1 1 0 1 2 1 3 A: 1 1 2 1 2 1 1 2 3 1 3 3 0 0 1 0 1 1 2 1 1 0 1 2 1 3
Q: 0 0 0 1 0 2 1 0 3 3 0 1 1 2 3 3 A: 0 0 0 1 0 2 1 0 3 3 0 1 1 2 3 3
Q: 3 0 3 3 1 A: 3 0 3 3 1
Q: 3 1 1 3 2 3 2 3 1 2 1 2 0 2 1 A: 3 1 1 3 2 3 2 3 1 2 1 2 0 2 1
Q: 0 0 1 3 2 1 2 2 3 2 0 3 2 1 3 0 1 3 2 3 3 3 2 A: 0 0 1 3 2 1 2 2 3 2 0 3 2 1 3 0 1 3 2 3 3 3 2
Q: 2 A: 2
Q: 2 3 A: 2 3
Q: 1 0 1 1 1 3 0 0 3 3 2 0 1 1 3 2 3 1 2 2 1 2 0 1 3 2 3 2 0 0 A: 1 0 1 1 1 3 0 0 3 3 2 0 1 1 3 2 3 1 2 2 1 2 0 1 3 2 3 2 0 0
Q: 0 3 0 3 1 2 2 0 3 2 0 2 0 0 2 1 2 2 0 2 A: 0 3 0 3 1 2 2 0 3 2 0 2 0 0 2 1 2 2 0 2
Q: 3 3 3 1 1 0 3 3 2 3 2 1 2 1 1 1 1 0 2 1 2 1 2 3 A: 3 3 3 1 1 0 3 3 2 3 2 1 2 1 1 1 1 0 2 1 2 1 2 3
Q: 3 1 2 0 3 2 1 0 3 1 2 2 2 1 1 3 0 1 3 2 0 1 A: 3 1 2 0 3 2 1 0 3 1 2 2 2 1 1 3 0 1 3 2 0 1
Q: 2 1 1 2 2 1 3 2 1 3 0 1 2 2 A: 2 1 1 2 2 1 3 2 1 3 0 1 2 2
Q: 2 2 1 1 1 3 0 0 2 1 3 0 2 3 3 3 A: 2 2 1 1 1 3 0 0 2 1 3 0 2 3 3 3
Q: 0 0 2 2 1 A: 0 0 2 2 1
Q: 2 0 0 3 2 1 0 1 2 2 0 2 1 3 2 2 0 0 2 2 1 1 A: 2 0 0 3 2 1 0 1 2 2 0 2 1 3 2 2 0 0 2 2 1 1
Q: 0 3 1 1 3 3 2 2 2 3 1 3 3 3 0 1 0 3 2 0 3 3 0 0 3 1 2 3 2 A: 0 3 1 1 3 3 2 2 2 3 1 3 3 3 0 1 0 3 2 0 3 3 0 0 3 1 2 3 2
Q: 0 2 1 3 3 3 0 0 1 1 3 1 2 3 0 0 1 1 3 0 0 2 0 0 A: 0 2 1 3 3 3 0 0 1 1 3 1 2 3 0 0 1 1 3 0 0 2 0 0
Q: 1 3 2 1 1 1 1 2 0 1 0 1 1 3 1 1 1 0 3 3 1 2 3 1 0 0 0 0 1 1 A:
```

Figure 6: An example of the input to the model to test in-context learning with an alphabet size of 4, 20 in-context samples, and evaluating on a sample with length 30.

We look to understand how the accuracy of in-context learning drops off as the alphabet size decreases. We also try a variant for the alphabet of size 8 where we perturb the distribution to force there to be at least one random case of adjacent repeats in each context (that is, two symbols next to each other would be identical).

Finally, we try a variant with index hints, where we place a unique integer index in between each token in the input sequence and mirror this in the output sequence. The indices are a contiguous, increasing set of integers - for each sample, they're initialized at a random value - i.e. the first index isn't always the same (this is so that we avoid memorizing just the indices corresponding to the length up to which we see in the in-context samples, and not generalizing when we see more indices at test-time when evaluating on longer sequences). An example input with index hints can be seen in 7.



```
Q: 24 3 25 2 26 2 27 2 28 2 29 3 30 3 A: 24 3 25 2 26 2 27 2 28 2 29 3 30 3
Q: 29 2 30 1 31 2 2 1 3 0 4 1 5 2 6 3 7 2 8 2 9 0 10 1 11 1 12 0 A: 29 2 30 1 31 2 2 1 3 0 4 1 5 2 6 3 7 2 8 2 9 0 10 1 11 1 12 0
Q: 14 0 15 3 16 3 17 2 18 3 19 0 20 0 21 1 22 2 23 2 24 3 25 2 26 1 A: 14 0 15 3 16 3 17 2 18 3 19 0 20 0 21 1 22 2 23 2 24 3 25 2 26 1
Q: 14 1 15 2 16 2 17 0 18 0 19 2 20 1 21 1 22 0 23 2 24 1 25 3 26 1 27 1 A: 14 1 15 2 16 2 17 0 18 0 19 2 20 1 21 1 22 0 23 2 24 1 25 3 26 1 27 1
Q: 27 3 28 0 29 0 30 2 31 1 2 3 3 3 4 2 5 1 6 2 7 3 8 1 9 1 10 A: 27 3 28 0 29 0 30 2 31 1 2 3 3 3 4 2 5 1 6 2 7 3 8 1 9 1 10
Q: 4 A: 4
Q: 5 0 6 2 7 0 8 2 9 1 10 1 11 1 12 0 13 3 14 2 A: 5 0 6 2 7 0 8 2 9 1 10 1 11 1 12 0 13 3 14 2
Q: 23 3 24 3 25 1 26 1 27 2 28 2 29 A: 23 3 24 3 25 1 26 1 27 2 28 2 29
Q: 16 2 17 0 18 0 19 0 20 3 21 1 22 2 23 0 24 2 25 0 26 A: 16 2 17 0 18 0 19 0 20 3 21 1 22 2 23 0 24 2 25 0 26
Q: 15 0 16 0 17 2 18 3 19 2 20 0 21 2 22 2 23 A: 15 0 16 0 17 2 18 3 19 2 20 0 21 2 22 2 23
Q: 2 2 3 1 4 2 5 3 6 1 7 0 8 1 9 2 10 1 11 0 12 0 A: 2 2 3 1 4 2 5 3 6 1 7 0 8 1 9 2 10 1 11 0 12 0
Q: 26 A: 26
Q: 24 3 25 0 26 2 27 0 28 0 29 3 30 1 31 0 2 2 3 3 4 1 5 1 6 A: 24 3 25 0 26 2 27 0 28 0 29 3 30 1 31 0 2 2 3 3 4 1 5 1 6
Q: 23 0 24 2 25 0 26 1 27 1 28 1 29 3 30 1 31 3 2 2 3 1 4 0 5 0 6 A: 23 0 24 2 25 0 26 1 27 1 28 1 29 3 30 1 31 3 2 2 3 1 4 0 5 0 6
Q: 5 3 6 1 7 1 8 3 9 2 10 2 11 3 12 0 13 0 14 1 15 3 16 0 17 A: 5 3 6 1 7 1 8 3 9 2 10 2 11 3 12 0 13 0 14 1 15 3 16 0 17
Q: 18 3 19 1 20 0 21 1 22 3 23 1 24 2 25 0 26 3 27 3 28 2 29 0 A: 18 3 19 1 20 0 21 1 22 3 23 1 24 2 25 0 26 3 27 3 28 2 29 0
Q: 23 2 24 3 25 0 26 1 27 1 28 1 29 1 30 3 31 0 2 1 3 1 4 3 5 0 6 0 A: 23 2 24 3 25 0 26 1 27 1 28 1 29 1 30 3 31 0 2 1 3 1 4 3 5 0 6 0
Q: 30 2 31 3 2 0 3 3 4 0 5 3 A: 30 2 31 3 2 0 3 3 4 0 5 3
Q: 6 2 7 3 8 2 9 0 10 1 11 A: 6 2 7 3 8 2 9 0 10 1 11
Q: 25 2 26 3 27 2 28 2 A: 25 2 26 3 27 2 28 2
Q: 15 0 16 1 17 1 18 2 19 3 20 3 21 2 22 1 23 1 24 3 25 3 26 1 27 0 28 0 29 3 A:
```

Figure 7: An example of the input to the model to test in-context learning with an alphabet size of 4, 20 in-context samples, and evaluating on a sample with length 30, with scratchpad.

5.2 Results & Analysis

We vary the alphabet sizes, test lengths, and other settings (perturbation, scratchpad) as described above and evaluate 10 samples at each setting. For each sample, we query the davinci version of GPT-3, which is a 175B parameter model, with the context and test question. We compute the exact match accuracy over the 10 samples for each setting.

We see from 8 that reducing the alphabet size (increasing repeats) indeed hurts length generalization. Increasing repeats in the form of the perturbation we described above also seems to mildly hurt length generalization. On the other hand, **adding index hints significantly improves in-context learning for the copy with repeats task** when the number of repeats is high (alphabet size 2). This suggests that the same techniques we used to enable length generalizability in small transformers might also work to improve generalizability in in-context learning, given that both are suspected to heavily leverage the mechanism of "induction heads".

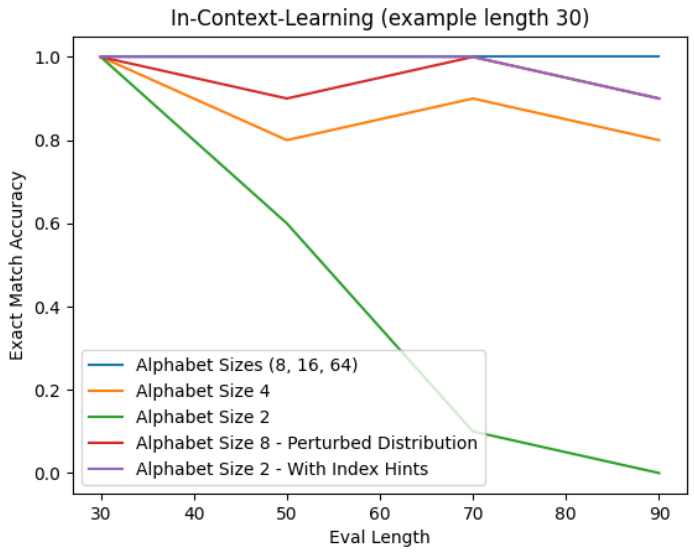


Figure 8: Exact match accuracy of model responses for the "copy with repeats" task with different alphabet sizes, settings, and evaluation lengths (number of examples in context and distribution of lengths for each context example held the same)

6 Conclusions and Future Work

In conclusion, we show three things:

1. Given individual tasks for which we can successfully length generalize, we can also successfully length generalize for a combination of these tasks in the multi-task setting, though the level of generalization may decrease particularly depending on the encoding used for the tasks.
2. We provide evidence that, combining the technique of scratchpads and index hints from Zhou et al. [2023] and Liu et al. [2023], transformers can show good length generalization on regular languages that they do not otherwise show (without modifications to the architecture).
3. The technique of applying index hints can help improve the ability of large models to perform and generalize during in-context learning as well, similar to how they help small models length generalize.

Future work could include the following:

1. **Length generalizability in the multi-task setting**: One potential direction here is to look at other methods of encoding tasks, including removing the task indicator altogether and

assessing if the model can learn to distinguish simple tasks from each other implicitly. It would also be interesting to look at more collections of tasks and understand if our results hold in general - for instance, one might create a library of RASP-L tasks and understand if we can learn effectively in the multi-task setting with any combination of them. One could also study how multi-task capabilities scale with model size - do we see that beyond a certain model size, different classes of tasks (potentially including non-RASP-L tasks) become amenable to length generalization by transformers? A mechanistic study of these small transformers might also yield more insight about whether the underlying reason they are able to learn some tasks well is because they truly learn algorithms related to the underlying RASP-L algorithm.

2. **Regular Languages vs RASP-L**: Can we extend this result and show length generalization on more regular languages? If so, can we then further study more complicated languages in the Chomsky hierarchy (e.g. context-free grammars)?
3. **In-Context Learning**: Can we apply the insights from index hints to more complicated ICL tasks than the "copy with repeats" task which showed promise?

References

Emmanuel Abbe, Samy Bengio, Aryo Lotfi, and Kevin Rizk. Generalization on the unseen, logic reasoning and degree curriculum, 2023.

Pranjal Awasthi and Anupam Gupta. Improving length-generalization in transformers via task hinting, 2023.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

Ta-Chung Chi, Ting-Han Fan, Alexander I. Rudnicky, and Peter J. Ramadge. Transformer working memory enables regular language reasoning and natural language length extrapolation, 2023.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy, 2023.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

Kevin Lacker. Giving gpt-3 a turing test. `https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html`, 2020. Accessed: 2023-12-13.

Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching arithmetic to small transformers, 2023.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Thomas McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability, 2023.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=De4FYqjFueZ`.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2022. URL `https://openreview.net/forum?id=iedYJm92o0a`.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads, 2022.

OpenAI. Gpt-4 technical report, 2023.

Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=R8sQPpGCv0`.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers, 2021.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length generalization, 2023.